(19) BUNDESREPUBLIK DEUTSCHLAND

DEUTSCHES PATENTAMT

(12) **Offenl gungsschrift**

(10) **DE 43 23 787 A 1**

(51) Int. Cl.[6]: G 06 F 11/28

(21) Aktenzeichen: P 43 23 787.8
(22) Anmeldetag: 15. 7. 93
(43) Offenlegungstag: 19. 1. 95

DE 43 23 787 A1

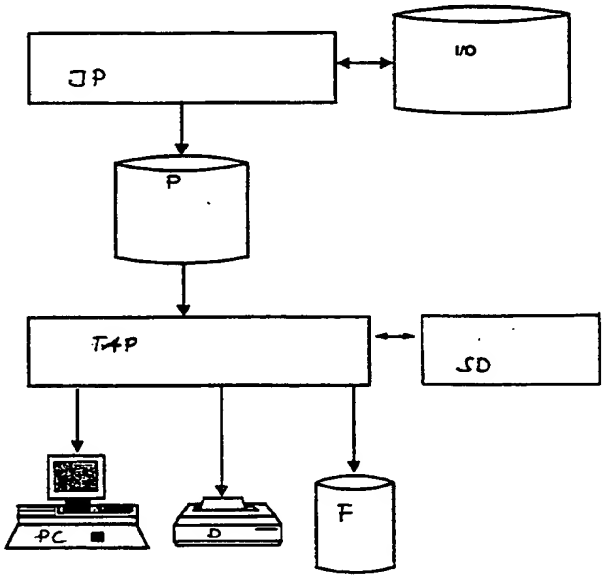
(71) Anmelder:
Siemens AG, 80333 München, DE

(72) Erfinder:
Jüttner, Peter, Dipl.-Inform., 81827 München, DE;
Kolb, Sebald, Dipl.-Math., 85521 Ottobrunn, DE;
Sieber, Stefan, Dipl.-Inform., 81739 München, DE;
Zimmerer, Peter, Dipl.-Inform., 81737 München, DE

Prüfungsantrag gem. § 44 PatG ist gestellt

(54) Verfahren zum Test eines objektorientierten Programms

(57) Mit der Erfindung wird ein Verfahren vorgestellt, das zum Test objektorientierter Programme diejenigen Programmstellen, welche von der Kommunikation der Objekte untereinander betroffen sind instrumentiert. Beim Ablauf des Programms werden sämtliche Kommunikationsvorgänge zwischen den Objekten mit zugehörigen Parametern in einer Protokolldatei aufgezeichnet. Diese Protokolldatei wird anschließend geeignet aufbereitet, so daß z. B. durch eine grafische Darstellung direkt die Ergebnisse des objektorientierten Designs mit dem Ablauf des Programms verglichen werden können.



DE 43 23 787 A1

Beschreibung

In vermehrtem Umfang hat sich bei der Erstellung umfangreicher Programme die Technik des objektorientierten Programmierens etabliert. Diese Technik bietet sowohl bei der Erstellung der Programme als auch beim Ablauf verschiedenste Vorteile.

Dem Programmierer wird beispielsweise das strukturierte Programmieren erleichtert. Weiterhin ist es einfach, Klassen aus bereits vorhandenen Programmen für neue Programme zu übernehmen. Da Objekte zur Laufzeit erzeugt und auch wieder gelöscht werden können, wird zu jedem Zeitpunkt des Ablaufs nur der aktuell benötigte Speicher beansprucht. Im Zusammenhang mit der Erstellung der objektorientierten Programme hat sich die Technik des OOD (object oriented design) [1] durchgesetzt. Diese Technik macht sich grafische Eingabemöglichkeiten bei der Erstellung von Programmen zunutze und ist in [2]—[5] näher erläutert.

Schwierigkeiten bestehen zur Zeit beim Test objektorientierter Programme. Dies liegt insbesondere an der dynamischen Struktur beim Ablauf dieser Programme. Da während des Programmablaufs Objekte erzeugt und wieder gelöscht werden, sowie Methoden aufgerufen werden, ist es schwierig, bei einem Test alle Objekte und Methodenaufrufe bezüglich ihrer Richtigkeit und ihrer Parameter zu überprüfen.

Verfahren und Werkzeuge, die einen umfangreichen Test von objektorientierten Programmen zulassen, sind derzeit nicht bekannt.

Der im folgenden erläuterten Erfindung liegt die Aufgabe zugrunde, ein Verfahren zum Test eines objektorientierten Programmes anzugeben.

Diese Aufgabe wird gemaß den Merkmalen des Patentanspruchs 1 gelöst.

Weiterbildungen der Erfindung ergeben sich aus den Unteransprüchen.

Da die Funktion objektorientierter Programme auf der Kommunikation der Objekte untereinander basiert, wird bei der Erfindung in vorteilhafter Weise die Kommunikation der Objekte untereinander analysiert. Dies geschieht erfindungsgemäß dadurch, daß alle kommunikationsrelevanten Stellen in den Programmen, d. h. in den Klassen und in den Methodenaufrufen dahingehend instrumentiert werden, daß die testrelevante Information (z. B. Methodenname, Parameterwerte, Aufrufer und gerufenes Objekt) beim beispielsweise Methodenaufruf oder beim Instantiieren bzw. Löschen eines Objekts in einem separaten Protokoll festgehalten werden können. Ein weiterer Bestandteil des erfindungsgemäßen Verfahrens besteht in der vorteilhaften Auswertung einer erstellten Protokolldatei. Hierdurch wird einem Testenden der Programmablauf übersichtlich dargestellt.

In vorteilhafter Weise geschieht die Veranschaulichung des Programmablaufs in grafischer Art. Wenn man diese Darstellung so wählt, daß sie der grafischen Repräsentation beim object oriented design entspricht, so kann man einfach manuell oder automatisch die Spezifikation des Programmes mit dem tatsächlichen Ablauf vergleichen.

Die erfindungsgemaß vorgesehene Instrumentierung für Methoden bietet den Vorteil, daß man beim Test leicht überprüfen kann, mit welchen Parametern welche Methode aufgerufen wurde. Weiterhin kann man statistisch auswerten, wie oft eine Methode aufgerufen wurde. Durch die Instrumentierung der Methodenaufrufe gewinnt man Informationen über die übergebenen Parameter und darüber, mit welchen Methodenaufrufen welche Methoden in welchen Objekten aufgerufen werden. Durch die Instrumentierung von beispielsweise den Methodenaufrufstellen in den verschiedenen Objekten gewinnt man Informationen, die man vor allen Dingen für die Analyse verketteter Methodenaufrufe benötigt. Verkettete Methodenaufrufe sind solche Methodenaufrufe, die beispielsweise von einem Objekt ausgehen und in einem weiteren Objekt wiederum einen Methodenaufruf bewirken und welcher dann in einem nächsten Objekt wieder eine Methode aufrufen kann.

Ein Vorteil des erfindungsgemäßen Verfahrens besteht vor allen Dingen in der Erstellung von Objektbilanzen. Damit läßt sich statistisch leicht feststellen, welche und wie viele Objekte von welchen Klassen während des Programmablaufs instantiiert wurden. Weiterhin kann man daraus ersehen, ob alle aufgerufenen Objekte auch wirklich benutzt wurden und ob mit den Methoden die richtigen Objekte angesprochen wurden. Man erkennt auch leicht, ob alle instantiierten Objekte auch wieder gelöscht worden sind.

Weiterhin gewinnt man so die Möglichkeit, die Tätigkeit des Programmierers überwachen zu können. Häufig gibt es in größeren Unternehmen an Programmierer die Anweisung, bereits erstellte Klassen oder Programmteile in neuen Programmen wieder zu verwenden. Mit dem erfindungsgemäßen Verfahren kann man nun leicht überprüfen, ob die Klassen, die in ein neues Programm eingebunden wurden, auch verwendet werden, oder ob sie nur einen quasi unnötigen Ballast im Programmcode darstellen.

Im folgenden wird die Erfindung anhand von Figuren und Beispielen weiter erläutert.

Fig. 1 zeigt als Beispiel ein Blockschaltbild des erfindungsgemäßen Verfahrens.

Fig. 2 zeigt die Spezifikation eines objektorientierten Programmes.

Fig. 3 zeigt die Realisierung des Programmes von **Fig.** 2.

Fig. 1 zeigt als Beispiel ein Blockschaltbild des erfindungsgemäßen Verfahrens. Mit I/O ist die Aus- und Eingabe des spezifizierten und nicht instrumentierten objektorientierten Programmes bezeichnet. Mit IP ist das instrumentierte objektorientierte Programm dargestellt. P bezeichnet die Protokolldatei, die beim Test erstellt wird. TAP stellt das Test-Auswerteprogramm dar, mit dem die beim Test erstellte Protokolldatei aufbereitet wird. SD heißen die Spezifikationsdaten, mit denen das objektorientierte Programm spezifiziert wurde. Über das Test-Auswerteprogramm TAP können die aufbereiteten Protokolldaten beispielsweise auf einem PC auf einem Drucker und in einer Datei F ausgegeben werden. In diesem Beispiel des erfindungsgemäßen Verfahrens geschieht beim Testen eines objektorientierten Programmes folgendes. Die Original- Eingaben, die I/O, werden dem instrumentierten objektorientierten Programm IP zugeführt. Beim Abarbeiten des instrumentierten Programmes IP werden die Eingabedaten I/O verarbeitet und falls im Programm IP instrumentierte Programmteile angesprochen werden, ergeben sich Testprotokoll-Daten, welche in einer Protokolldatei P oder einem Protokoll P abgelegt werden. Die Ausgabe-Daten von IP entsprechen dabei den Ausgabe-Daten des nicht instrumentier-

ten Programmes. Beispielsweise sind in dem objektorientierten Programm Methoden und Aufrufstellen von Methoden in Objekten instrumentiert. Insbesondere können bei in C++ erstellten Programmen beispielsweise alle Memberfunktionen, statische Memberfunktionen, Konstruktoren, Destruktoren, überladene Operatoren, Friends und Casts instrumentiert sein.

Anschließend wird das Protokoll P beispielsweise durch das Test-Auswerteprogramm TAP ausgewertet. Im Zuge der Auswertung kann die Protokolldatei beispielsweise grafisch dargestellt werden. Eine besonders geeignete Form der grafischen Darstellung bietet jenes grafische Format, welches beim Spezifizieren des Programmes im Zuge des objektorientierten Designs verwendet wurde. Man kann so leicht im Vergleich von Protokoll-Daten mit den Spezifikationsdaten SD Unterschiede zwischen Programm und zugehöriger Spezifikation feststellen. Es besteht die Möglichkeit, diese Unterschiede am Bildschirm oder am Drucker auszugeben oder auch in einer Datei abzuspeichern, um sie weiterverarbeiten zu können. Bei der Analyse des Protokolls P durch das Test-Auswerteprogramm TAP sind auch noch andere Auswertemethoden denkbar. Beispielsweise kann die Objektlandschaft angezeigt werden. Das bedeutet, zu einem aktuellen Zeitpunkt oder für einen Zeitabschnitt können alle existierenden Objekte ausgegeben werden. Weiterhin ist es denkbar, einen Methodenaufrufbaum darzustellen. Dieser Methodenaufrufbaum zeigt alle weiteren Methodenaufrufe an, die ein selektierter Methodenaufruf zur Folge hat. Weiterhin kann man sich denken, die Objektlandschaft mit Aufrufbeziehung anzeigen zu lassen, wobei alle jemals existenten Objekte einschließlich ihrer gegenseitigen Methodenaufrufe angezeigt werden können. Zusätzlich ist es denkbar, die Klassenlandschaft mit ihren Aufrufbeziehungen darzustellen. Dies hat zur Folge, daß alle existierenden Klassen einschließlich ihrer Methodenaufruf-Beziehungen untereinander gezeigt werden. Alle diese vorab aufgezählten Darstellungen werden in den verschiedenen OOD-Methoden zur Spezifikation eines objektorientierten Programmes verwendet. In analoger Weise kann man die Protokolldatei auf statistische Merkmale hin analysieren. Beispielsweise können alle Objekte gezählt werden. Es könn n sämtliche Methodenaufrufe gezählt werden, und man kann über eine Analyse der maximal und minimal existierenden Objekte bzw. Methodenaufrufe Aussagen über die Programm-Komplexität machen.

Fig. 2 zeigt ein objektorientiertes Programm in grafischer Darstellung nach Art des objektorientierten Designs (OOD). Es sind drei Objekte o1 bis o3 aus zugehörigen Klassen K1 bis K3 dargestellt. o1 ruft beim Objekt o2 eine Methode ma mit den Parametern x und y auf. o1 ruft auch beim Objekt o3 eine Methode mb mit den Parametern z, x und c auf. **Fig. 2** zeigt hier beispielsweise die Spezifikationsform des objektorientierten Programmes. Für diese Spezifikation folgt nun der Source-Code des objektorientierten Programmes.

1. C++-Source-Code

Im folgenden wird das Beispiel in C++-Syntax beschrieben.

```
class K1 {
        public:

                void m() {

                        K2 o2;                          // standard constructor call
                        K3 o3;                          // standard constructor call
                        int     x = 1,
                                y = 2;
                        float   z = 1.5;
                        char    c = '*';

                        ...
                        // other statements of the method
                        o2.ma(x, y);
                        ...
                        // other statements of the method
                        o3.mb(z, x, c);
                        ...
                        // other statements of the method

                }
```

```
}

class K2 {
        public:
                void ma(int p, int q) {
                        ...
                        // statements of the method
                }
}

class K3 {
        public:
                void mb(float l, int m, char n) {
                        ...
                        // statements of the method
                }
}

void main() {
        K1 o1;          // standard constructor call
        ...
                        // other statements
        o1.m();
        ...
                        // other statements
}
```

Für das gezeigte Beispiel folgt nun der nach dem erfindungsgemäßen Verfahren instrumentierte Source-Code. Darin sind die instrumentierten Stellen durch Fettdruck hervorgehoben.

2. Instrumentierter C++-Source-Code

```
class K1 {
        public:

                K1()
                {                       // standard constructor was added
                        protfile << "call K1::K1()\n"
                                 << "in file ... at line ... happened\n";
                        protfile << "parameters:\n";
                        protfile << "no\n";
                }

                void m() {
                        protfile << "call K1::m()\n"
                                 << "in file ... at line ... happened\n";
                        protfile << "parameters:\n";
                        protfile << "no\n";
                        K2 o2;                  // standard constructor call
                        protfile << "dcl K2 o2 in K1::m()\n"
                                 << "in file ... at line ...\n";
                        K3 o3;          // standard constructor call
                        protfile << "dcl K3 o3 in K1::m()\n"
                                 << "in file ... at line ...\n";
                        int x = 1,
                            y = 2;
                        float z = 1.5;
                        char c = 'a';

                        ...
                        // other statements of the method
                        protfile << "call o2.ma(x, y) from class K2\n"
                                 << "in K1::m() in file ... at line ... started\n";
                        protfile << "parameters:\n";
                        protfile << " int x = " << x << ", int y = " << y << "\n";
                        o2.ma(x, y);
                        protfile << "call o2.ma(x, y) from class K2\n"
                                 << "in K1::m() in file ... at line ... ended\n";
                        protfile << "parameters:\n";
                        protfile << " int x = " << x << ", int y = " << y << "\n";
                        ...
```

```
                    // other statements of the method
                    protfile << "call o3.mb(z, x, c) from class K3\n"
                            << "in K1::m() in file ... at line ... started\n";
                    protfile << "parameters:\n";
                    protfile << " float z = " << z << ", int x = " << x
                            << ", char c = " << c << "\n";
                    o3.mb(z, x, c);
                    protfile << "call o3.mb(z, x, c) from class K3\n" }
                            << "in K1::m() in file ... at line ... ended\n";
                    protfile << "parameters:\n";
                    protfile << " float z = " << z << ", int x = " << x
                            << ", char c = " << c << "\n";
                    ...
                    // other statements of the method

            }

    }

    class K2 {
            public:

                    K2()
                    {                                   // standard constructor was added
                            protfile << "call K2::K2()\n"
                                    << "in file ... at line ... happened\n";
                            protfile << "parameters:\n";
                            protfile << "no\n";
                    }

                    void ma(int p, int q) {
                            protfile << "call K2::ma(p,q)\n"
                                    << "in file ... at line ... happened\n";
                            protfile << "parameters at the begin of K2::ma(p,q):\n";
                            protfile << " int p = " << p << ", int q = " << q << "\n";
                            ...
                            // statements of the method
                            protfile << "parameters at the end of K2::ma(p,q):\n";
                            protfile << " int p = " << p << ", int q = " << q << "\n";
                    }
    }
```

```
class K3 {
        public:

                K3()
                {                       // standard constructor was added
                        protfile << "call K3::K3()\n"
                                 << "in file ... at line ... happened\n";
                        protfile << "parameters:\n";
                        protfile << "no\n";
                }

                void mb(float l, int m, char n) {
                        protfile << "call K3::mb(l, m, n)\n"
                                 << "in file ... at line ... happened\n";
                        protfile << "parameters at the begin of K3::mb(l, m, n):\n";
                        protfile << " float l = " << l << ", int m = " << m
                                 << ", char n = " << c << "\n";
                        ...
                        // statements of the method
                        protfile << "parameters at the end of K3::mb(l, m, n):\n";
                        protfile << " float l = " << l << ", int m = " << m
                                 << ", char n = " << c << "\n";
                }
}


void main() {

        {ofstream protfile("protocol");          // protocol file}
        protfile << "call main()\n"
                 << "in file ... at line ... happened\n";
        protfile << "parameters:\n";
        protfile << "no\n";
        K1 o1;                                   // standard constructor call
        protfile << "dcl K1 o1 in main()\n"
                 << "in file ... at line ...\n";

        ...
                // other statements
        protfile << "call o1.m() from class K1\n"
                 << "in main() in file ... at line ... started\n";
        protfile << "parameters:\n";
```

```
protfile << "no\n";
o1.m();
protfile << "call o1.m() from class K1\n"
        << "in main() in file ... at line ... ended\n";
protfile << "parameters:\n";
protfile << "no\n";
...
                    // other statements
}
```

Es ist hier zwar der Source-Code instrumentiert, doch ist es auch möglich, wenn man genügend Kenntnis über die eingesetzte Hardware hat und den entsprechenden Compiler in Verbindung mit dem Objektcode, der für das objektorientierte Programm erzeugt wird, gut kennt, daß die entsprechende Instrumentierung auch auf der Objektcode-Ebene erfolgt.

Hier ist nur die Instrumentierung des Source-Codes gewählt worden, um ein anschauliches Beispiel darstellen zu können. Dieser instrumentierte Source-Code führt zu einer Ausgabeprotokolldatei, die im Anschluß folgt.

3. Ausgabe der Protokolldatei

Die beim Ablauf des Programmes (main function) erstellte Protokolldatei ist im folgenden aufgelistet. Die eingefügten Leerzeilen dienen ausschließlich der besseren Lesbarkeit.

Die angegebenen Parameterwerte sind natürlich nur exemplarisch, sie hängen logischerweise von den hier nicht im einzelnen aufgeführten Statements in den Methodenrümpfen ab. Ebenso sind bei der Ausgabe von "in file ... at line ..." die angegebenen drei Punkte ("...") jeweils durch den aktuellen Dateinamen und durch die aktuelle Zeilennummer im (nicht instrumentierten) Source-Code ersetzt.

```
call main()
in file ... at line ... happened
parameters:
no

call K1::K1()
in file ... at line ... happened
parameters:
no

dcl K1 o1 in main()
in file ... at line ...

call o1.m() from class K1
in main() in file ... at line ... started
parameters:
no

call K1::m()
in file ... at line ... happened
parameters:
no

call K2::K2()
in file ... at line ... happened
parameters:
no

dcl K2 o2 in K1::m()
in file ... at line ...

call K3::K3()
in file ... at line ... happened
parameters:
no

dcl K3 o3 in K1::m()
in file ... at line ...

call o2.ma(x, y) from class K2
in K1::m() in file ... at line ... started
parameters:
```

```
int x = 1, int y = 2

call K2::ma(p, q)
in file ... at line ... happened
parameters at the begin of K2::ma(p, q):
int p = 1, int q = 2

parameters at the end of K2::ma(p, q):
int p = 1, int q = 2

call o2.ma(x, y) from class K2
in K1::m() in file ... at line ... ended
parameters:
int x = 1, int y = 2

call o3.mb(z, x, c) from class K3
in K1::m() in file ... at line ... started
parameters:
float z = 1.5, int x = 1, char c = *

call K3::mb(l, m, n)
in file ... at line ... happened
parameters at the begin of K3::mb(l, m, n):
float l = 1.5, int m = 1, char n = *

parameters at the end of K3::mb(l, m, n):
float l = 1.5, int m = 1, char n = *

call o3.mb(z, x, c) from class K3
in K1::m() in file ... at line ... ended
parameters:
float z = 1.5, int x = 1, char c = *

call o1.m() from class K1
in main() in file ... at line ... ended
parameters:
no
```

Mit dem erfindungsgemäßen Verfahren wird nun über die Aufbereitung durch das Test-Auswerteprogramm TAP die grafische Darstellung von Fig. 3 erzeugt. Sie zeigt drei Objekte o1, o2 und o4, wobei o1 am Objekt o2 die Methode ma mit den Parametern x und y aufruft und am Objekt o4 die Methode mb mit z, x und c aufruft. Hier ist deutlich zu erkennen, daß Fig. 3 im Vergleich zur Spezifikation, welche in Fig. 2 dargestellt ist, eine Abweichung aufweist.

In Fig. 3 wird die Methode mb nämlich am Objekt o4, anstatt am Objekt o3 aufgerufen, was nicht mit dem Design übereinstimmt. Analog dazu können beispielsweise weitere Abweichungen vom Design, wie z. B. fehlende oder zusätzliche (und damit eventuelle überflüssige) Methodenaufrufe ermittelt werden.

Wichtig ist es, beim erfindungsgemäßen Verfahren zu beachten, daß die durch die Instrumentierung des Source-Codes beispielsweise ermittelte Protokolldatei nur als Mittel zum Zweck dient. Wesentlich ist es, daß die Protokolldatei weiterverarbeitet wird und geeignet beispielsweise grafisch aufbereitet wird. Geeignet bedeutet in diesem Zusammenhang, daß die Aufbereitung in Affinität zur Darstellung in der Phase des objektorientierten Designs erfolgt. Ein Ziel ist es hierb i beispielsweise, den eigentlichen Programmablauf anschließend direkt mit dem Design, d. h. mit den Ergebnissen aus der OOD-Phase vergleichen zu können.

Insbesondere kann die Instrumentierung des zu testenden Programmes auch vorteilhaft automatisiert durch ein hier nicht näher beschriebenes Instrumentierungs-Programm erfolgen.

Literaturangeaben:

[1] Introduction to Object Technol gie Steve Cook OOP 1992, Conference Proceedings SIGS Publicati ns, New York, 1992.
[2] H. Heilmann, A. Gebauer, M. Simon: Objektorientiertes Software Engineering. Handbuch der modernen Datenverarbeitung (HMD), 30. Jg., 1993, Heft 170, S. 11 – 23.
[3] S. Schäfer: Klassische Entwurfstechniken für die objektorientierte Softwareentwicklung. Handbuch der modernen Datenverarbeitung (HMD), 30. Jg., 1993, Heft 170, S. 47 – 54.
[4] H. Heß, A.-W. Scheer: Methodenvergleich zum objektorientierten Design von Softwaresystemen. Handbuch der modernen Datenverarbeitung (HMD), 29. Jg., 1992, Heft 165, S. 117 – 137
[5] H. Schaschinger, H. Sikora, I. Bäuchler: Objektorientierte Analyse- und Designmethoden – Überblick und kritische Betrachtung. Softwaretechnik-Trends, Band 11, Heft 4, Nov. 1991, S. 32 – 43.

Patentansprüche

Verfahren zum Test eines objektorientierten Programmes (o1, o2, o3, o4),

a) bei dem mindestens ein Kommunikations-Vorgang (ma(x,y)) zwischen Objekten des Programmes in einem Protokoll (P) protokolliert wird, indem mindestens ein Programmteil des Programmes, welcher vom Kommunikationsvorgang betroffen ist, instrumentiert wird,

b) und bei dem der Test (TAP) dadurch erfolgt, daß das Protokoll (P) ausgewertet wird.

2. Verfahren nach Anspruch 1, bei dem das Protokoll (P) automatisch in jenem grafischen Format aufbereitet wird, in welchem die Spezifikation des objektorientierten Programms erfolgte, so daß das Programm gegen die Spezifikation getestet werden kann.

3. Verfahren nach einem der Ansprüche 1 oder 2, bei dem Methoden (ma(x,y),mb(z,x,c)) instrumentiert werden.

4. Verfahren nach einem der vorhergehenden Ansprüche, bei dem Methodenaufrufe instrumentiert werden.

5. Verfahren nach einem der obigen Ansprüche, bei dem Bilanzen erstellt werden.

6. Verfahren nach Anspruch 6, bei dem Bilanzen über Klassen, oder Objekte, oder Methoden erstellt werden.

7. Verfahren nach einem der obigen Ansprüche, bei dem über eine statistische Auswertung die Wiederverwendung mindestens einer in dem objektorientierten Programm eingefügten Klasse überprüft wird.

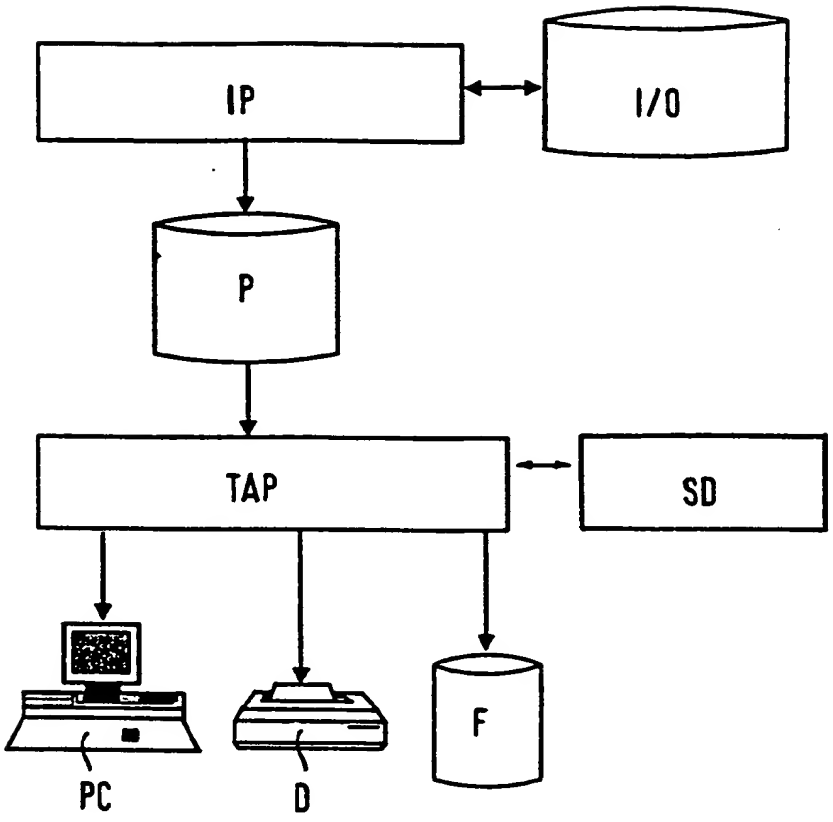
Hierzu 2 Seite(n) Zeichnungen

– Leerseite –

Num
Int. Cl.[6]:
Offenlegungstag:

DE 43 23 787 A1
G 06 F 11/28
19. Januar 1995

FIG 1



Nummer: DE 43 23 787 A1
Int. Cl.[6]: G 06 F 11/28
Offenlegungstag: 19. Januar 1995

FIG 2

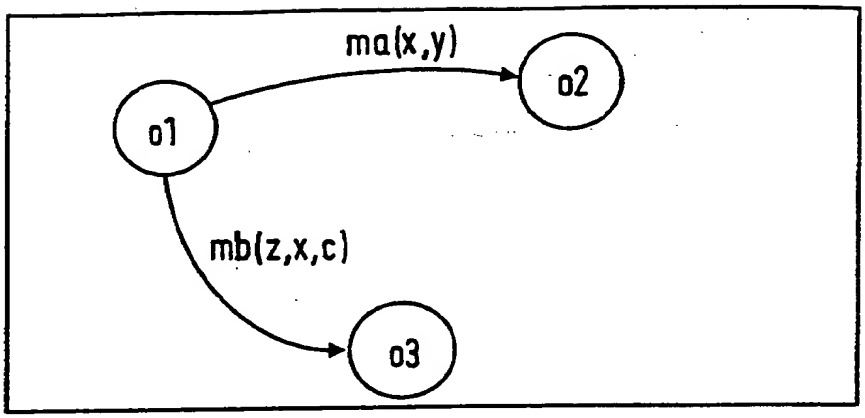


FIG 3